An Algebraic Specification of a Model Factory, Part III

Alban Ponse
Emile Verschuren
Programming Research Group, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
Email: alban@fwi.uva.nl

Abstract

This paper reports on a case study of a model factory. In the papers [VW92, HP92] this factory is described and simulated in the specification language PSF. The present case study is based on these papers, and describes a specification of some features of memory management.

Key Words & Phrases: Specification Language, PSF, ADT (Abstract Data Types), Process Algebra.
1985 Mathematics Subject Classification: 68Q40.
1987 CR Categories: D.1.3, D.2.1, F.1.2.

1 Introduction

The work reported on in this paper is part of a project, the overall aim of which is to give an algebraic specification of a model factory. The actual model factory, and therefore the context of this project is provided by the Amsterdam research site of DIGITAL EQUIPMENT CORPORATION (DEC) called COOPERATIVE ENGINEERING CENTER(CEC)[1].

The purpose of this project is twofold. A first goal is to apply the particular specification language PSF [MV89, Mau90, MV90] in a modelling on a relatively large scale. The model factory built for and programmed at CEC suits this goal perfectly. Secondly, it is intended to investigate whether desired features can be easily specified and simulated on a symbolic level with PSF. Such an effort would then facilitate an analysis of desired features. Two previous PSF specifications of the model factory that form part of this project are described in [VW92, HP92], respectively. The second one of these forms the basis for the present paper, and is listed in an Appendix. The idea is that the cooperation between DEC CEC and the University of Amsterdam will result in a series of PSF specifications, each of which extends the previous one and will be made by a fresh team (as to guarantee easy accessibility).

In this paper we investigate the possibilities of specifying in PSF a flexible interface between *high-level control* (= which machine must/can perform what mechanical activity next; how should the flow of information be directed) and *low-level device control* (= associated to machines: what mechanical task is to be performed, and - when fulfilled - what message is to be sent). While in the specification of [HP92] 'overall control' is based on synchronisation between these two levels of control, we here define it according to (some aspects of) "BASEstar", an integration platform developed at DEC's. Roughly speaking, synchronisation of either party is now relaxed to synchronisation with BASEstar. The database features of BASEstar take care of the necessary, asynchronous updating. We conclude that PSF is only to a certain extent suitable for analysis of such flexible interfaces. Notably, also real-time constraints, not yet specifiable in PSF, play a prominent rôle in such analysis.

[1]DEC CEC, Einsteingebouw, Kabelweg 21, 1014 BA Amsterdam, The Netherlands, (+31 20 5866000)

Acknowledgements. We would like to thank Ronald van Riessen, manager at DEC CEC, for making this project feasible. Ruud Hakkesteegt also played an important role in explaining the merits and shortcomings of the present implementation of the model factory. Section 2, a general introduction to PSF and its use for specifying CIM architectures, is based on previous texts of Willem Jan Fokkink, Sjouke Mauw and Jos van Wamel. Finally we acknowledge Jan Bergstra for starting up this project.

Contents

2 The specification language PSF

The specification language PSF, a Process Specification Formalism, is developed at the University of Amsterdam. It is a combination of an algebraic specification formalism for data, and process algebra. PSF therefore has an explicit data-part and a process-part using the data. Also various tools exist. Some general references to PSF are [MV89], [Mau90], [MV90] and [Vel91]. In this section PSF and its use for the specification and verification of CIM-Architectures are shortly discussed.

2.1 Data in PSF

This section contains a brief introduction to the specification of data by algebraic specifications, as is prescribed in PSF. More detailed background information can be found in [BHK89] and the references cited there, in particular in the survey paper [MG85].

Data are time-independent objects with mathematical properties such as symmetries, invariants and representation results. In PSF, data are constructed from a given, finite number of *constants* and *functions*. Furthermore, each data term is typed by a sort name (such as Naturals or Booleans). Properties of data can be defined by *equations* of the form $d = d'$, where d and d' are data terms of the same sort that may contain (typed) variables.

For example let Nat be the sort of the naturals, with a constant 0 of type Nat and a function $S : \mathrm{Nat} \to \mathrm{Nat}$, denoting the successor function while no equations are specified. Then the *initial algebra* (representation of all closed terms modulo provable equality) of Nat is the collection

$$\{\ 0,\ S(0),\ S(S(0)),\ \ldots\ \}.$$

One can add a function such as $+ : \mathrm{Nat} \times \mathrm{Nat} \to \mathrm{Nat}$, denoting addition, with equations

$$
\begin{aligned}
&n + 0 = n \\
&n + S(m) = S(n + m)
\end{aligned}
$$

where m, n are variables of the type Nat. In this case the initial algebra is

$$
\begin{array}{ll}
\{ & [\ 0,\ 0+0,\ (0+0)+0,\ \ldots\ ] \\
 & [\ S(0),\ 0+S(0),\ S(0)+0,\ \ldots\ ] \\
 & [\ S(S(0)),\ S(0)+S(0),\ 0+S(S(0)),\ \ldots\ ] \\
 & \cdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}
\end{array}
$$

where the collections between square brackets are the equivalence classes of the sort Nat generated by provable equality (a congruence relation).

In general, it can be hard to decide whether or not two closed data terms are equal. A means for doing this is offered by the theory of *term rewriting systems.* One can try to set up the equations for the data in such a way that by applying them from left to right only, each reduction of a closed data term is finite. This means that in every reduction ends with a term, called a *normal form*, to which none of the (left-hand sides of the) equations matches anymore. The aim is to have *unique* normal forms, i.e., if two closed data terms are equal, their normal forms are literally the same. Then it clearly is decidable for each pair of closed data terms whether or not they are equal, by simply reducing them to normal forms.

In the example of the natural numbers above, the normal forms are the expressions of the form $S(S(\ldots S(0))\ldots)$. Thus the closed data terms of the sort Nat have unique normal forms. However, if an equality $m + n = n + m$ is added to the equations for the $+$, one gets infinite reductions such as $0 + S(0) = S(0) + 0 = 0 + S(0) = \ldots$.

Modules enable the explicit declaration of algebraic data types as described above. A module consists of two parts: one in which the *algebraic signature* is given and a second which contains the *equational logic.* So a data module is a structure of the following form:

```
data module Naturals
begin

  < algebraic signature >

  < equational logic >

end
```

Signatures are sets of declarations of *sorts* and *functions.* A *constant* is simply a function with no arguments. For example, the algebraic signature of the module Naturals is as follows:

```
    sorts
      NAT

    functions
      0   :             -> NAT
      S   : NAT         -> NAT
      _+_ : NAT # NAT -> NAT
```

In the second part the equations for the data terms are given. In the case of the Naturals this is:

```
    variables
      m,n :       -> NAT
```

```
equations
  [N1]  n+0 = n
  [N2]  n+S(m) = S(n+m)
```

An important feature is *import*. A module can import other modules, whence an initial algebra that has been created in a first module can be used in a second one.

2.2 Processes in PSF

In this section a short description of process algebra is given, because it forms, together with module algebra and term rewriting techniques, the algebraic basis of PSF. After a short description of process algebra, some characteristics of PSF are mentioned, and some of the tools that are available for the manipulation of specifications in PSF.

A process can be described informally as *the behaviour of a system* over time. It turns out, however, to be an intricate philosophical problem to describe the difference between processes and data in a formal way. This is especially true when processes are parameterised with data and complex interactions between processes and data take place. In [Pon92] attention is paid to process algebras with data.

Process algebra is a formal approach to the specification and verification of concurrent systems. It may be considered the study of concurrent communicating processes in an algebraic framework. Several process algebras have been developed in the past decade, but here we discuss only ACP, the Algebra of Communicating Processes. There are different variants of ACP. For instance, abstract or untimed ACP [BW90] and Real Time Process Algebra [BB91]. We restrict ourselves to abstract ACP, which forms part of the core of PSF. Other approaches to process algebra can be found in [Hoa85], [Hen88] and [Mil89].

We shortly introduce ACP. The theory of ACP has two parameters: a set of A *actions* and a partial communication function $\gamma : A \times A \rightarrow A$ that is commutative and associative. Actions are the most basic *processes* considered, and are assumed to be not subject to further division. Three important binary operators to compose processes into more complex ones are sequential composition ($\cdot$), alternative composition or choice ($+$), and parallel composition ($\|$). Partcularly, $p \parallel q$ represents the process that either performs a first action of p, or of q, or a 'communication action' of p and q, i.e. the simultaneous occurrence of a first action of p and one of q. There are also special constants in process algebra, the most important of which are the *deadlock* (δ) and the *silent step* (τ). If a system can perform no action but a deadlock, it is blocked: it cannot continue or terminate successfully. The silent step denotes an *internal* action, i.e., an action that is hidden (not observable). Two system operators must also be mentioned here: *encapsulation* and *hiding*. Encapsulation forces the components of a parallel system to communicate and thus to synchronise. Hiding renames all internal actions of a system into a silent step. Once the signature of a process algebra is defined, an axiom system can be defined. An axiomatic *verification* of a system is a formal proof of the fact that the observable behaviour of the system (abstracting from internal actions) satisfies a given requirement. Algebraic verification can be performed simply by manipulating equations, using the axioms of a given algebra. Another option is verification in some specific *model* for processes (a process algebra). Well-known models for processes are graphs or labelled transition systems (after dividing out some behavioural congruence such as *bisimulation*). Practical examples of specifications and verifications of concurrent systems can be found in the literature. Examples of such systems are protocols in computer networks, systolic algorithms, and logistic protocols in factories. Some references are [Bae90] and [vWa92].

The formal specification language PSF is based on abstract ACP and PSF was developed in order to provide ACP with the means to write tools for the verification and simulation of specifications. The formal definition of PSF also makes it possible to verify whether a specification is syntactically correct or not. The definition of PSF can be found in [MV90].

A PSF specification is organised in *data modules* and *process modules*. The different modules of a specification are connected via an import-export mechanism. The *exports* section of a module contains the visible part of the algebraic signature of a module when it is imported into another module via its *imports* section.

In process modules the processes of a specification are specified. For example: if a system SYS consisting of a sender S and a receiver R is specified, S can be put in one process module and R in another. Then the process SYS can be specified in a *top-module*, which means that it forms the highest level in the specification in which all other modules are imported.

In data modules the abstract data types used in a process specification are defined. No standard types (*sorts* in PSF) such as naturals or reals are available. This part of PSF is not based on process algebra but on module algebra and term rewriting techniques, as described in the previous section. In process modules data may be imported and used in various manners to specify processes: communication can be defined in data parametric way; processes can be defined recursively in a data parametric way with tests on data values; and there is a 'sum' operator for defining all choices between the closed data-instantiations of a parametric process.

The specification language PSF is equipped with several tools for the analysis of a specification. We mention some of these that are typical for analysis by simulation:

1. The *term rewriter*. By means of this tool expressions are rewritten to normal forms, which is useful to verify whether or not functions have been defined correctly.

2. The *initial*-tool, which makes it possible to calculate (part of) the initial algebra of a specification. It calculates the different elements of a given sort (type) using the functions and the rewrite rules defined on these functions.

3. The *simulator*. This tool makes a stepwise analysis of a specification possible. The simulator performs successive actions, which the user chooses from a menu. To put it more technically: with the simulator traces through the *state-space* of a system can be generated. The simulator can also make random runs. This option yields, in general, a fast method to detect deadlocks or other undesired system behaviour.

Tools 1 and 2 are suitable for the analysis of the data in specifications.

Some recent examples of PSF specifications can be found in [Bru91] and [?]. For more detailed information on the syntax of PSF the reader is referred to [MV90].

2.3 Specification and Verification of CIM-Architectures in PSF

Computer Integrated Manufacturing, shortly CIM, refers to industrial production in a setting in which computers are used. Both the product flow and the information flow can first be designed and analysed by means of computers. Next, also the actual control of industrial production can be organised by means of computers. In such production often a lot of machines operate independently of each other. Providing a correct functioning of the total of all machines, computers and transport services is not a trivial exercise. Before such a CIM architecture can be build, one must design it. Such a design, when judged 'correct', describes a properly functioning system on a symbolic level. In particular, the possibilities to use process algebra as a means for the specification and verification of CIM architectures are manifold. Clearly, PSF offers a suitable environment for the specification of such architectures.

Globally speaking, a plant can be seen as a composition of several concurrently operating *Workstations*. Every Workstation is responsible for some well-defined part of the manufacturing process. Workstations are connected to each other via some transport service, managing various conveyor belts. This cooperation is controlled by one supervising process, being able to communicate with all physical components of the plant. This can be achieved by providing each Workstation by a local *Controller*. In turn these Controllers can communicate with (the relevant) units of the transport service. Optionally, such a conceptually and physically natural part of the plant can be specified as a *Control Zone*.

Finally, the overall control of production and information flow can be specified on a top level (see also [Mau90]).

The specification of Workstations and transport units can be given in PSF. When abstracting from the internal actions, it is possible to determine its external, or *observable* behaviour. At the high level view on the flow of products, one can be only interested in how products pass the Workstations. Also, at the high level view on the flow of information, one may only be interested in the commands given to a Workstation or unit of transport, and how these relate to the so called *request flow*, the information representing *what* should be manufactured.

Finally, it must be remarked that no real time features are captured. Consequently, important features like *efficiency* (maximal productivity of machines) and *tuning* (synchronisation of the speed of the machines) cannot be modelled. Currently, these features have been added to process algebra [BB91].

3 The model factory

In Section 3.1 we describe the model factory as it is modelled in [HP92]; the PSF code of that specification is listed in the Appendix. Some aspects of the simplification of the actual model factory are discussed in 3.2. In Section 3.3 we mention some characteristics of our modelling. This section contains no PSF syntax.

3.1 Overall behaviour

The model factory produces *Printed Circuit Boards* (PCBs). A PCB is an electrical device consisting of a (print-)*board* with a maximum of six electronic *components* placed upon it. Three kinds of PCBs are manufactured. These are called the *product types* A, B and C. The product types differ on the kind of board and on the number and kind of components used. There are two kinds of boards, called *board1* and *board2*, and three kinds of components, referred to as *red*, *yellow* and *green*. Because a PCB does not have to contain exactly six components, a fourth component, *emp*(ty) is introduced. Initially, a PCB is just an empty board.

During manufacturing, one, two or three PCBs are operated on at the same time. Such a cluster is called a *batch*. The PCBs in a batch stay (conceptually) together during the entire production process and must be of the same product type.

Figure 1 shows the track each batch will traverse. The figure is split up in eight *Control Zones* (CZs), separated by the dashed lines. A CZ is a conceptual unit that connects all manufacturing, storing and transporting units in the model factory to the appropriate local *Controller*. The overall control of the factory is specified as the concurrent execution of all CZs, and is explained in more detail in [HP92]. Now we describe the manufacturing of batches step by step, using the CZs merely for reference to locations. In Figure 1 the different types of transporting units are numbered as to give easy reference to the figures and PSF code occurring later in the paper.

CZ-1. The *Raw Material Stores* (RMSs) are stores containing empty boards; each RMS contains boards of one type. The empty boards are put from the RMSs onto the conveyor belt by board movers, and are further operated on as a batch.

CZ-2. Here the batch passes the *Screen Printing Station* (SP). At this station solder paste is put on the boards.

CZ-3 and CZ-4. Now the components can be placed. Some components are placed by *Placement Station 1* (CP1) and others by *Placement Station 2* (CP2). A batch for production of type A has to pass both stations CP1 and CP2, a batch for B must only pass CP1 and a batch for product type C skips CP1, but has to pass CP2 twice. Therefore C-batches have to do the whole tour twice.

CZ-5. After the components have been placed each batch passes the *Reflow and Clean Station* (RC) and then goes into an *In Process Store* (IPS). There are three IPSs, one for each product-type. The PCBs stay here until a request comes in at the local Controller. Then the requested number of PCBs of the requested product type are taken from the IPS and transported further as a batch.

CZ-6 and CZ-7. Here the batch passes the *Test and Repair Station* (TR). This station performs a visual test (are all components well-placed?) and an electronic test. If anything is wrong the faulty PCB(s) is (are) repaired and the entire batch must go through the loop in zones CZ-6 and CZ-7 to be tested again. If finally everything is okay, the batch is further transported.

CZ-8. If the production of the batch is completed, it is sent to the *Warehouse*. A C-batch that is only halfway (it has to go through the Placement Station CP2 for the second time) passes the Warehouse and is put in a buffer, the *Second Side Buffer* (SSB). From there the C-batch can be put on the conveyor belt again by a board mover.

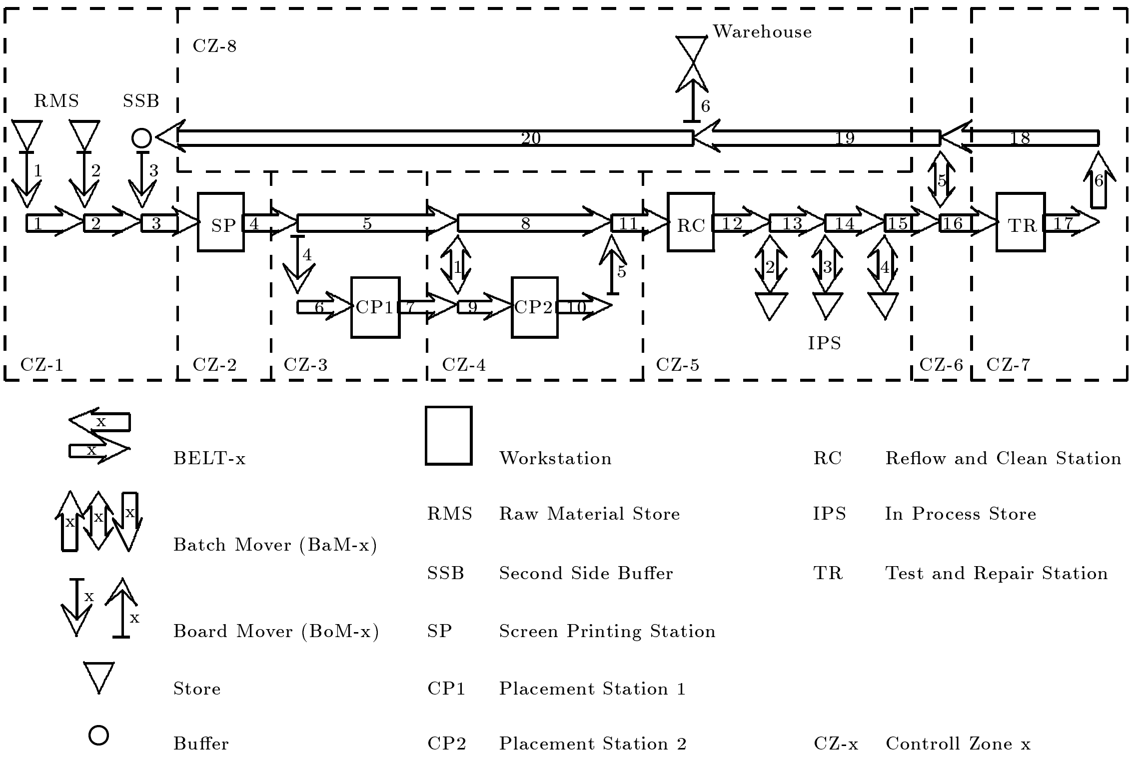


Figure 1: Flow of Products in the model factory

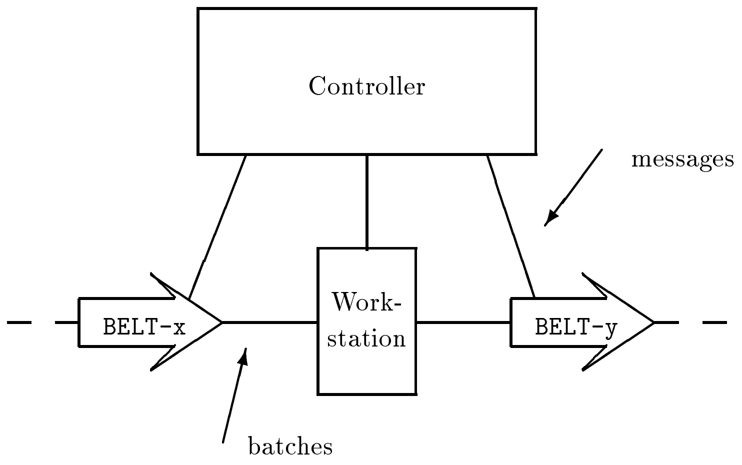
3.2 Simplifications

The model factory as described above is a simplified representation of the one at DEC CEC. We neither considered the supply of accessories (components and empty trays for components and boards), nor the shipping of the final products from the Warehouse. The RMSs and the Warehouse are simply assumed to be of a looping, respectively infinite capacity. Another simplification concerns the Test and Repair Station TR. We assume that a batch is always first judged faulty and repaired, and when visiting TR for the second time is judged `ok`. In reality however, batches are repaired manually in a special buffer and then put back in the production process. Here also accessories can be needed, which then have to be supplied.

3.3 Modelling of the control structure

In [VW92], a PSF specification of the model factory is given. In that specification the product types, boards, components, PCBs and batches are defined, as well as a number of primary processes such as conveyor belts, movers and a number of Workstations. The control of the manufacturing process is implicitly specified. This specification models the production in batches that can contain exactly three PCBs, reflecting the current functionality of the model factory at DEC CEC.

In the future, the model factory is meant to be able to deal with batches of size one and two as well. In the PSF specification in [HP92], which is based on that from [VW92], this feature has been added. However, the main characteristic of the specification described there is its distributed control structure. In this specification all ports that connect the manufacturing and transporting units of the model factory are used for the communication of batches. These are the *batch-ports*. Messages are all communications with a Controller on special *data-ports*. A (local) Controller can then pass the messages through to another part of the model factory. This situation can be depicted as follows:



In the current paper we try to model control in a way that is intended to correlate with an integration platform developed at DEC CEC, called BASEstar. Roughly speaking, one of the functionalities of BASEstar is to provide a flexible interface between high-level control (software) and low-level device control. With respect to the model factory, this means that control of the physical units can be specified in Controllers while abstracting from the actual duration a unit takes to perform (part of) its task, or from the fact that it can only *once* signal its current state to its Controller. So compared with the previous specification in [HP92], synchronisation between Controllers and low-level devices is relaxed to synchronisation of any of these with BASEstar. The database features of BASEstar take care of the necessary 'asynchronous updating'. The modelling of these memory management aspects of BASEstar is discussed in the next section.

4 Modelling memory management aspects of BASEstar

In the PSF specification of the model factory in [HP92] on which our work is based, the Workstations and the Controllers directly interchanged data. In that view the send and receive actions of the Controllers and Workstations are synchronised: messages are communicated directly between a Controller and a Workstation, and requests are communicated directly between Controllers.

In this report we view a more complicated situation. The data transactions are carried out via BASEstar, an integration platform that serves as an intermediary for data exchanged by a Controller with its 'environment'. The behaviour of the data transactions by means of BASEstar is specified in the modules `BaseProcess` and `BaseMemory` which are specified below.

4.1 The data module BaseMemory

We decided to specify some core, but rather characteristic, features of BASEstar in a separate data module `BaseMemory`. In this module we specify the memory of BASEstar as a data structure, together with the basic functions `read`, `write` and `update`. Firstly the memory of BASEstar consists of a set of `POINTS`. Each point represents an virtual entity of which the value can be read and written. The main notion of the memory management of BASEstar is then the separation into a 'read-part' and a 'write-part' of each point. A point consists of two actual parts of memory, each of which can contain a value. When a value is assigned to a point, BASEstar writes the value in just one part of it, the write-part. The value of the other part of the point remains unchanged. However, BASEstar returns the contents of this other part when the point is read. In an update, BASEstar replaces the values of the read-part of a point by that of its write-part. The function `reset` resets the values in both the read and write-part of a point to the value 'reset'. The function `init` generates a set of `POINTS` with initial values. The `Point` and `Points` functions are simply constructors of the data type `POINTS`. We model the memory of BASEstar as a set of points each of which can contain the two values discussed above and an identifier. To close up with the previous specification of the model factory, these identifiers are the port names defined in that specification. The PSF code of `BaseMemory` is as follows (where ... abbreviates patterns that are nearly identical):

```
data module BaseMemory
begin
  exports
    begin
      sorts
        POINTS, POINT
      functions
        read:   POINTS # DATA-PORT             -> MESSAGE
        write:  POINTS # DATA-PORT # MESSAGE    -> POINTS
        reset:  POINTS # DATA-PORT             -> POINTS
        update: POINTS                         -> POINTS
        init:                                  -> POINTS
        init:   DATA-PORT                      -> POINTS
        Point:  DATA-PORT # MESSAGE # MESSAGE   -> POINT
        Points: POINTS # POINTS # POINTS # POINTS #
                POINTS # POINTS # POINTS # POINTS # POINTS -> POINTS
        Points: POINT # POINTS                 -> POINTS
        Points: POINT                          -> POINTS
    end
  imports
    Ports,Messages
  variables
    n,n1,n2    : -> FIN
    ps,ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq: -> POINTS
    p:          ->      POINT
    m,m1,m2,m3: ->      MESSAGE
    r:          ->      REQUEST
```

```
    dp,dp1,dp2,dp3:->   DATA-PORT
  equations
    [Bm1]  read(Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq),dp(1,n)) = read(ps1,dp(1,n))
    [Bm2]  read(Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq),dp(2,n)) = read(ps2,dp(2,n))
    .....  ..dp(3,n)..dp(4,n)..dp(5,n)..dp(6,n)..dp(7,n)..
    [Bm8]  read(Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq),dp(8,n)) = read(ps8,dp(8,n))
    [Bm9]  read(Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq),req(n)) = read(psreq,req(n))
    [Bm10] read(Points(Point(dp,m1,m2),ps),dp) = m1
    [Bm11] read(Points(Point(dp1,m1,m2),ps),dp2) = read(ps,dp2) when follows(dp2,dp1) = true
    [Bm12] read(Points(Point(dp1,m1,m2)),dp2) = m1

    [Bm13] reset(Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq), dp(1,n)) =
                 Points(reset(ps1,dp(1,n)),ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq)
    [Bm14] reset(Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq), dp(2,n)) =
                 Points(ps1,reset(ps2,dp(2,n)),ps3,ps4,ps5,ps6,ps7,ps8,psreq)
    .....  ..dp(3,n)..dp(4,n)..dp(5,n)..dp(6,n)..dp(7,n)..
    [Bm20] reset(Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq), dp(8,n)) =
                 Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,reset(ps8,dp(8,n)),psreq)
    [Bm21] reset(Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq),req(n)) =
                 Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,reset(psreq,req(n)))
    [Bm22] reset(Points(Point(dp,m1,m2),ps),dp) = Points(Point(dp,reset,reset),ps)
    [Bm23] reset(Points(Point(dp1,m1,m2),ps),dp2) = Points(Point(dp1,m1,m2),reset(ps,dp2))
                              when follows(dp2,dp1) = true
    [Bm24] reset(Points(Point(dp1,m1,m2)),dp2) = Points(Point(dp1,reset,reset))

    [Bm25] write(Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq), dp(1,n),m) =
                 Points(write(ps1,dp(1,n),m),ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq)
    [Bm26] write(Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq), dp(2,n),m) =
                 Points(ps1,write(ps2,dp(2,n),m),ps3,ps4,ps5,ps6,ps7,ps8,psreq)
    .....  ..dp(3,n)..dp(4,n)..dp(5,n)..dp(6,n)..dp(7,n)..
    [Bm32] write(Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq), dp(8,n),m) =
                 Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,write(ps8,dp(8,n),m),psreq)
    [Bm33] write(Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq),req(n),m) =
                 Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,write(psreq,req(n),m))
    [Bm34] write(Points(Point(dp,m1,m2),ps),dp,m3) =    Points(Point(dp,m1,m3),ps)
    [Bm35] write(Points(Point(dp1,m1,m2),ps),dp2,m3) = Points(Point(dp1,m1,m2),write(ps,dp2,m3))
                              when follows(dp2,dp1) = true
    [Bm36] write(Points(Point(dp1,m1,m2)),dp2,m3) = Points(Point(dp1,m1,m3))

    [Bm37] init = Points(init(first(1)),init(first(2)), init(first(3)),init(first(4)),
                         init(first(5)),init(first(6)), init(first(7)),init(first(8)),
                           init(req(0)) )
    [Bm38] init(dp(n1,n2)) = Points(Point(dp(n1,n2),av,av),init(dp(n1,n2+1)))
                                      when lt(n2+1,ports(n1)) = true
    [Bm39] init(dp(n1,n2)) = Points(Point(dp(n1,n2),av,av)) when n2+1 = ports(n1)
    [Bm40] init(req(n)) = Points(Point(req(n),av,av),init(req(n+1)))
                                      when lt(n+1,reqs) = true
    [Bm41] init(req(n)) = Points(Point(req(n),av,av)) when n+1 = reqs

    [Bm42] update(Points(ps1,ps2,ps3,ps4,ps5,ps6,ps7,ps8,psreq)) =
               Points(update(ps1),update(ps2),update(ps3),update(ps4),
                      update(ps5),update(ps6),update(ps7),update(ps8),
                      update(psreq) )
    [Bm43] update(Points(Point(dp,m1,m2),ps)) = Points(Point(dp,m2,m2),update(ps))
    [Bm44] update(Points(Point(dp,m1,m2))) = Points(Point(dp,m2,m2))
end BaseMemory
```

4.2 The process module BaseProcess

Messages are sent from an origin to a destination. In the current implementation of the model factory at DEC's, a message is sent from its origin to BASEstar, which in turn sends the message to its destination. When at BASEstar a message arrives, BASEstar determines the point belonging to this message and puts it in there. After this has happened, BASEstar can receive other messages. However, BASEstar cannot send a message immediately after is has been received: it first has to update its memory. The mechanism by which BASEstar passes messages on is called *polling*. The two parts of memory contained in a point are used for this. When BASEstar receives a message, it writes the contents of this message into the write-part of the determined point; when BASEstar sends a message the value of this message is the contents of the read-part of the point from which that message is sent. A separate process in BASEstar periodically updates the read-parts with the write parts of all points. We model this behaviour of BASEstar in a process module `BaseProcess`. The process `Poll` causes BASEstar to update its memory periodically. The process `Base` models the behaviour of the memory. After reading a point, `Base` resets the point in order to prevent itself from reading the same message again and again. To address messages and points the port names are used, and the `rm` and `sm` actions can now easily be connected with the ones of the previous specification in [HP92]. In order to distinguish the 'directions' of messages, we introduce constants `CB`, `BC`, `SB` and `BS`. These have the following meaning: when a Controller sends a message to `BaseProcess`, this message is marked with `CB`; a message from `BaseProcess` to a Controller with `BC`; from a Station or Belt to `BaseProcess` with `SB` and vice versa with `BS`. The specification of the process module `BaseProcess` is quite straightforward:

```
process module BaseProcess
begin
  exports
    begin
      atoms
        rpol,spol,pol
      processes
        BaseStar
    end
  imports
    Communication, Process-ids, Ports, BaseMemory, Message-set, Data-port-set
  processes
    Base: POINTS
    Poll
  sets
    of atoms
      U = {rpol,spol}
  communications
    rpol | spol = pol
  variables
    ps:         -> POINTS
    m:          -> MESSAGE
  definitions
    Base(ps)
      = rpol . Base(update(ps)) +
        sum (dp in DATA-PORT-SET,
          sm(dp, BS, read(ps,dp)) . Base(reset(ps,dp)) +
          sm(dp, BC, read(ps,dp)) . Base(reset(ps,dp)) +
          sum (m in MESSAGE-SET,
            rm(dp,SB,m) . Base(write(ps,dp,m)) +
            rm(dp,CB,m) . Base(write(ps,dp,m))) )
    Poll = spol . Poll
    Baseprocess = encaps(U,Base (init) || Poll)
end BaseProcess
```

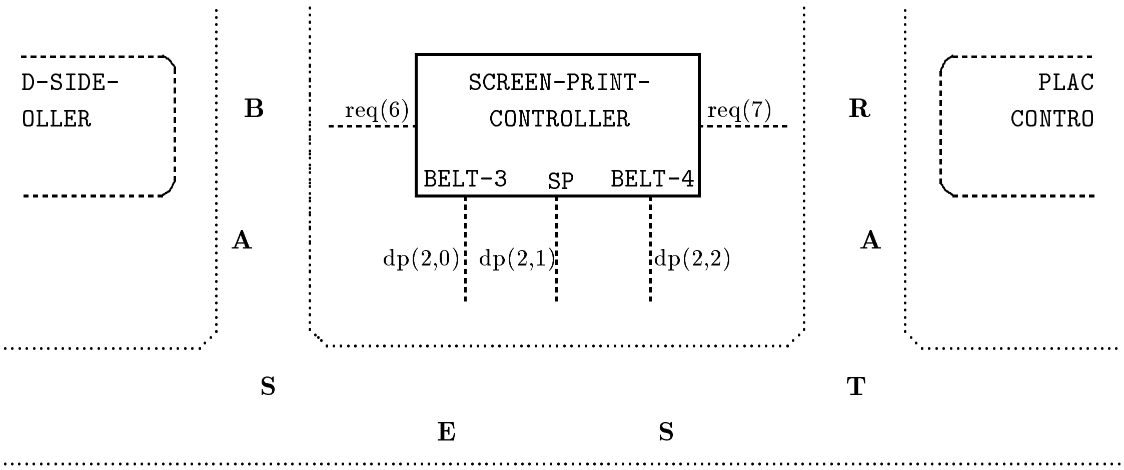
5 A case study

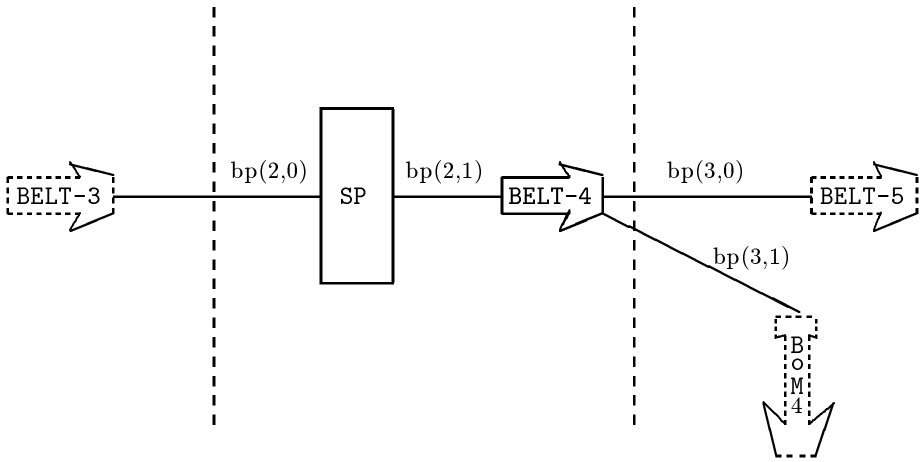
To study the modelling of the aspects of memory management of BASEstar discussed in the previous section, we extend part of the specification given in [HP92] with the specification of `BaseProcess`.

In order to simulate this extension, we define a 'simulation environment' for one Control Zone. The section is concluded with a trace obtained by simulation of our extension.

5.1 Specifications

For simplicity, we decided to extend the most simple Control Zone, i.e., CZ-2 (see page 7) with `BaseProcess`. This extension of CZ-2 can be depicted as follows (cf. [HP92]):





For each unit in CZ-2, i.e., the Screen Printing station `SP` and the belt `BELT-4`, a sub-control-process is defined. Furthermore there is one sub-control-process that propagates incoming requests. The `SCREEN-PRINT-CONTROLLER` has all these sub-control processes running in parallel, thus increasing flexibility. Basically, its functionality is to organise control for its own units. Finally Control Zone 2 (CZ-2) is defined by the parallel execution of the Controller and its units. We refer to [HP92] for a detailed description of CZ-2, and more generally for a description of the modelling of control, and here only focus on some adaptations made to facilitate a case study.

To perform our case study, we introduce a testing process for CZ-2. This process is called `ZONE2` and defined in a process module `Zone2` (see below). The functionality of `ZONE2` is to simulate the CZ-2 part of the PSF specification in [HP92] *in cooperation with* `BaseProcess`. The adaptation necessary for our case study, and indeed for the way we model the mentioned aspects of BASEstar, can be summarised as follows:

- Port names are parameterised over a (finite) data-type. This is done to facilitate the updating character of `BaseProcess`, and this can easily be extended to the whole specification in [HP92].
- The data type `MESSAGES` is extended with 'location values' CB, ... and a new function `reset` as explained in the previous section.

Moreover, by nature of the limited modelling power of only CZ-2, the flow of requests, a global phenomenon, cannot be simulated properly on the level of CZ-2. This is the case by definition, and it may be reflected in the request flow that can be extracted from simulations. It is finally assumed that the updating triggered by the polling is *faster than* the request flow, so that requests cannot be overwritten before being read. This assumption seems quite reasonable.

We only give the top-level specification of our test process `ZONE2`. In this specification the processes `SEND` and `RECEIVE` simulate the communication of CZ-2 with the other CZs.

```
process module Zone2
begin
  exports
    begin
      processes
        ZONE2
    end
  imports
    Sender, Controll-zone-2, Receiver, Basestar, Data-port-set
  sets
    of BATCH-PORT
      BP = {bp(2,0),bp(3,0),bp(3,1)}
    of DATA-PORT
      DP = {dp(2,0),dp(2,1),dp(2,2),dp(3,0),req(6),req(7)}
    of atoms
      AT =   {rb(bp, b), sb(bp, b) | bp in BP, b in BATCH-SET}
           + {rm(dp, pe, m), sm(dp, pe, m) |
dp in DP, m in MESSAGE-SET, pe in PORT-EXT-SET}
    definitions
      ZONE2 = encaps(AT, SEND || CZ-2 || RECEIVE || BaseStar)
end Zone2
```

5.2 A simulation

In this section we provide a simulation trace obtained from `ZONE2`. We display a trace that contains no requests and that typically shows how the polling of `BaseProcess` affects the communications between the units in CZ-2. For example, a communication between `BELT-3` and the `SCREEN-PRINT-CONTROLLER` is split by `BaseProcess` into the first and the fourth communication actions (labelled by `com.`) below.

```
com. cm(dp(2, 0), SB, ho(id(C, bn(3), half)))
com. pol
com. cm(dp(2, 1), SB, av)
com. cm(dp(2, 0), BC, ho(id(C, bn(3), half)))
com. pol
com. cm(dp(2, 1), BC, av)
com. cm(dp(2, 1), CB, go)
com. pol
com. cm(dp(2, 2), SB, av)
com. cm(dp(2, 1), BS, go)
```

```
com. pol
com. pol
com. cb(bp(2, 0), batch(rc(pcb(board2, yellow, emp, yellow, emp, yellow, yellow)), bn(3)))
com. cm(dp(2, 0), SB, ho(id(B, bn(1), empty)))
com. pol
com. cm(dp(2, 0), BC, ho(id(B, bn(1), empty)))
atom print-screen(id(C, bn(3), half))
com. cm(dp(2, 1), SB, ho(id(C, bn(3), half)))
com. pol
com. cm(dp(2, 1), BC, ho(id(C, bn(3), half)))
com. cm(dp(2, 2), BC, av)
com. cm(dp(2, 2), CB, go-mm)
```

6 Conclusions

The work described here leads us to the conclusion that features of memory management of BASEstar can be specified in PSF. These features describe the polling mechanism that underlies BASEstar while abstracting from real time constraints. Consequently, we can only assume that the following constraint on this polling mechanism is satisfied: the updating triggered by the polling is *faster than* the request flow, so that requests cannot be overwritten before being read (cf. Section 5.1). Our specification of the above features is based on the PSF description of the model factory in the previous paper [HP92], the PSF code of which is listed in the Appendix. As to ease the PSF specification of the polling mechanism, some minor modifications in the PSF description of the model factory come in handy (see also Section 5.1):

1. Port names are parameterised over a (finite) data-type. This is done to define the updating character of BASEstar in a uniform way, and can easily be done for the whole specification in [HP92].

2. The data type `MESSAGES` used for defining the control of the model factory is extended with 'location values' CB, ... and a new function `reset`.

The present work shows that it is possible to model aspects of memory management on a symbolic level. This may perhaps be of use in the development of (products such as) BASEstar. However, in the end BASEstar is meant to serve as an integration platform in developing, maintaining and (possibly) adapting the actual software control of the model factory (or other applications), and therefore its presence/functionality need not play a rôle in symbolic analysis of the specification of software control.

References

[BB91] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Journal of Formal Aspects of Computing Science*, 3(2):142–188, 1991.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[Bae90] J.C.M. Baeten, editor. *Applications of Process Algebra.* Cambridge Tracts in Theoretical Computer Science 17. Cambridge University Press, 1990.

[Bru91] J. Brunekreef. A formal specification of two simple protocols for local area net works. Report P9109, University of Amsterdam, 1991.

[HP92] J.A. Hillebrand and A. Ponse. An algebraic specification of a Model Factory, part II. Report P9214, University of Amsterdam, 1992.

[Hen88] M. Hennessy. *Algebraic theory of processes.* MIT Press, Cambridge, Massachusetts, 1988.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[MG85] J. Meseguer and J.A. Goguen. Initiality, induction and computability. In M. Nivat and J.C Reynolds, editors, *Algebraic methods in semantics*, pages 459–541. Cambridge University Press, 1985.

[MV89] S. Mauw and G.J. Veltink. An introduction to PSF_d. In J. Díaz and F. Orejas, editors, *Proceedings TAPSOFT 89, Vol. 2*, volume 352 of *Lecture Notes in Computer Science*, pages 272–285. Springer-Verlag, 1989.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[Mau90] S. Mauw. Process algebra as a tool for the specification and verification of CIM–architectures. In J.C.M. Baeten, editor, *Applications of process algebra*, Cambridge Tracts in Theoretical Computer Science 17, pages 53–80. Cambridge University Press, 1990.

[Mil89] R. Milner. *Communication and concurrency.* Prentice-Hall International, 1989.

[Pon92] A. Ponse. *Process algebra with data.* PhD thesis, University of Amsterdam, 1992.

[VW92] S.F.M. van Vlijmen and A. van Waveren. An algebraic specification of a Model Factory. Report P9209, University of Amsterdam, 1992.

[Vel91] G.J. Veltink. The PSF toolkit. Report P9107, University of Amsterdam, 1991.

[vWa92] J.J. van Wamel. A study of a one bit sliding window protocol. Report P9212, Programming Research Group, University of Amsterdam, 1992.

Appendix

We finally list the specification defined in [HP92] on the following twelve pages. The overall control of production and information flow is specified in the last (top-)module `Model-factory`, defining the parallel execution of all control zones. The names in the specification refer to those used in Section 3. Further comments on this specification can be found in [HP92].

```
data module Naturals
begin
  exports
    begin
      sorts
        NAT
      functions
        0:                ->   NAT
        s:    NAT         ->   NAT
        add:  NAT # NAT   ->   NAT
        sub:  NAT # NAT   ->   NAT
    end
  variables
    x, y:   ->   NAT
  equations
    [N0]  add(x, 0) = x
    [N1]  add(x, s(y)) = s(add(x, y))
    [N2]  sub(x, 0) = x
    [N3]  sub(0, y) = 0
    [N4]  sub(s(x), s(y)) = sub(x, y)
end Naturals


data module Batch-sizes
begin
  exports
    begin
      sorts
        BATCH-SIZE
      functions
        zero:                ->   BATCH-SIZE
        one:                 ->   BATCH-SIZE
        two:                 ->   BATCH-SIZE
        three:               ->   BATCH-SIZE
        bn:    BATCH-SIZE    ->   NAT
        nb:    NAT           ->   BATCH-SIZE
    end
  imports
    Naturals
  variables
    x:   ->   NAT
  equations
    [BS0] bn(zero) = 0
    [BS1] bn(one) = s(0)
    [BS2] bn(two) = s(s(0))
    [BS3] bn(three) = s(s(s(0)))
    [BS4] nb(0) = zero
    [BS5] nb(s(0)) = one
    [BS6] nb(s(s(0))) = two
    [BS7] nb(s(s(s(0)))) = three
    [BS8] nb(s(s(s(s(x))))) = nb(s(s(s(x))))
end Batch-sizes


data module Loop-values
begin
  exports
    begin
      functions
        top:   ->   NAT
    end
  imports
    Naturals
  equations
    [I0] top = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))))))))
end Loop-values


process module Loop-set
begin
  exports
    begin
      sets of NAT
        LOOP-SET = NAT \ {add(n, s(top)) | n in NAT}
    end
  imports Loop-values
end Loop-set


data module Product-types
begin
  exports
    begin
      sorts
        PRODUCT-TYPE
      functions
        A:   ->   PRODUCT-TYPE
        B:   ->   PRODUCT-TYPE
        C:   ->   PRODUCT-TYPE
    end
end Product-types


data module Boards
begin
  exports
    begin
      sorts
        BOARD-TYPE
      functions
        board1:   ->   BOARD-TYPE
        board2:   ->   BOARD-TYPE
    end
end Boards
```

```
data module Components
begin
  exports
    begin
      sorts
        COMPONENT
      functions
        red:      ->   COMPONENT
        yellow:   ->   COMPONENT
        green:    ->   COMPONENT
        emp:      ->   COMPONENT
    end
end Components
```

```
data module Printed-circuit-boards
begin
  exports
    begin
      sorts
        PCB
      functions
        pcb:     BOARD-TYPE #
                 COMPONENT # COMPONENT # COMPONENT #
                 COMPONENT # COMPONENT # COMPONENT   ->   PCB
        B-to-be: PCB                                 ->   PCB
        C-to-be: PCB                                 ->   PCB
        sp:      PCB                                 ->   PCB
        rc:      PCB                                 ->   PCB
      end
  imports
    Boards, Components
end Printed-circuit-boards
```

```
data module Pcb-prototypes
begin
  exports
    begin
      functions
        pcb-empty:   PRODUCT-TYPE   ->   PCB
        pcb-half:    PRODUCT-TYPE   ->   PCB
        pcb-fin:     PRODUCT-TYPE   ->   PCB
    end
  imports
    Product-types, Printed-circuit-boards
  equations
    [P0] pcb-empty(A) = pcb(board1, emp, emp, emp, emp, emp, emp)
    [P1] pcb-empty(B) = B-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))
    [P2] pcb-empty(C) = C-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))
    [P3] pcb-half(A) = pcb(board1, yellow, red, emp, red, emp, yellow)
    [P4] pcb-half(C) = pcb(board2, yellow, emp, yellow, emp, yellow, yellow)
    [P5] pcb-fin(A) = pcb(board1, yellow, red, green, red, green, yellow)
    [P6] pcb-fin(B) = pcb(board2, yellow, emp, yellow, red, red, red)
    [P7] pcb-fin(C) = pcb(board2, yellow, emp, yellow, green, yellow, yellow)
end Pcb-prototypes
```

```
data module Batches
begin
  exports
    begin
      sorts
        BATCH
      functions
        batch: PCB   #   BATCH-SIZE   ->   BATCH
        del:   BATCH                  ->   BATCH
        rep:   BATCH                  ->   BATCH
        ok:    BATCH                  ->   BATCH
        size:  BATCH                  ->   BATCH-SIZE
    end
  imports
    Printed-circuit-boards, Batch-sizes
  variables
    pcb:   ->   PCB
    bs:    ->   BATCH-SIZE
    b:     ->   BATCH
  equations
    [B0] size(batch(pcb, bs)) = bs
    [B1] size(del(b)) = size(b)
    [B2] size(rep(b)) = size(b)
    [B3] size(ok(b)) = size(b)
end Batches
```

```
data module Batch-status
begin
  exports
    begin
      sorts
        STATUS
      functions
        empty:   ->   STATUS
        half:    ->   STATUS
        fin:     ->   STATUS
        del:     ->   STATUS
        rep:     ->   STATUS
        ok:      ->   STATUS
    end
end Batch-status
```

```
data module Batch-ids
begin
  exports
    begin
      sorts
        BATCH-ID
      functions
        id:  PRODUCT-TYPE # BATCH-SIZE # STATUS   ->   BATCH-ID
        bid: BATCH                                ->   BATCH-ID
    end
  imports
    Pcb-prototypes, Batches, Batch-status
  functions
    prod: PCB   ->   PRODUCT-TYPE
    stat: PCB   ->   STATUS
  variables
    pcb:   ->   PCB
    bs:    ->   BATCH-SIZE
    pt:    ->   PRODUCT-TYPE
  equations
    [Bi0]  bid(batch(pcb, bs)) = id(prod(pcb), bs, stat(pcb))
    [Bi1]  bid(del(batch(pcb, bs))) = id(prod(pcb), bs, del)
    [Bi2]  bid(rep(batch(pcb, bs))) = id(prod(pcb), bs, rep)
    [Bi2]  bid(ok(batch(pcb, bs))) = id(prod(pcb), bs, ok)
    [Bi3]  prod(pcb(board1, emp, emp, emp, emp, emp, emp)) = A
    [Bi4]  prod(B-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))) = B
    [Bi5]  prod(C-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))) = C
    [Bi6]  prod(pcb(board1, yellow, red, emp, red, emp, yellow)) = A
    [Bi8]  prod(pcb(board2, yellow, emp, yellow, emp, yellow, yellow)) = C
    [Bi9]  prod(pcb(board1, yellow, red, green, red, green, yellow)) = A
    [Bi10] prod(pcb(board2, yellow, emp, yellow, red, red, red)) = B
    [Bi11] prod(pcb(board2, yellow, emp, yellow, green, yellow, yellow)) = C
    [Bi12] prod(sp(pcb)) = prod(pcb)
    [Bi13] prod(rc(pcb)) = prod(pcb)
    [Bi14] stat(pcb(board1, emp, emp, emp, emp, emp, emp)) = empty
    [Bi15] stat(B-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))) = empty
    [Bi16] stat(C-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))) = empty
    [Bi17] stat(pcb(board1, yellow, red, emp, red, emp, yellow)) = half
    [Bi18] stat(pcb(board2, yellow, emp, yellow, emp, yellow, yellow)) = half
    [Bi20] stat(pcb(board1, yellow, red, green, red, green, yellow)) = fin
    [Bi21] stat(pcb(board2, yellow, emp, yellow, red, red, red)) = fin
    [Bi22] stat(pcb(board2, yellow, emp, yellow, green, yellow, yellow)) = fin
    [Bi23] stat(sp(pcb)) = stat(pcb)
    [Bi24] stat(rc(pcb)) = stat(pcb)
end Batch-ids
```

```
data module Requests
begin
  exports
    begin
      sorts
        ORDER, REQUEST
      functions
        ord: PRODUCT-TYPE # NAT          ->   ORDER
        req: PRODUCT-TYPE # BATCH-SIZE   ->   REQUEST
    end
  imports
    Product-types, Batch-sizes
end Requests
```

```
data module Messages
begin
  exports
    begin
      sorts
        MESSAGE
      functions
        reset:              ->   MESSAGE
        av:                 ->   MESSAGE
        ho:    BATCH-ID     ->   MESSAGE
        go:                 ->   MESSAGE
        go-up:              ->   MESSAGE
        go-dn:              ->   MESSAGE
        go-mm:              ->   MESSAGE
        go-ms:              ->   MESSAGE
        go-sm:              ->   MESSAGE
        go-ss:              ->   MESSAGE
        pres:  NAT          ->   MESSAGE
        take:  REQUEST      ->   MESSAGE
        give:  REQUEST      ->   MESSAGE
    end
  imports
    Batch-ids, Requests
end Messages
```

```
data module Batch-operations
begin
  exports
    begin
      functions
        screen-print: BATCH      ->   BATCH
        comp-place-1: BATCH      ->   BATCH
        comp-place-2: BATCH      ->   BATCH
        reflow-clean: BATCH      ->   BATCH
        test-repair:  BATCH      ->   BATCH
        cp1:          PCB        ->   PCB
        cp2:          PCB        ->   PCB
    end
  imports
    Batches, Pcb-prototypes
  variables
    pcb:   ->   PCB
    bs:    ->   BATCH-SIZE
  equations
    [Bo0]  screen-print(batch(pcb, bs)) = batch(sp(pcb), bs)
    [Bo1]  comp-place-1(batch(pcb, bs)) = batch(cp1(pcb), bs)
    [Bo2]  comp-place-2(batch(pcb, bs)) = batch(cp2(pcb), bs)
    [Bo3]  reflow-clean(batch(pcb, bs)) = batch(rc(pcb), bs)
    [Bo4]  test-repair(del(batch(pcb, bs))) = rep(batch(pcb, bs))
    [Bo5]  test-repair(rep(batch(pcb, bs))) = ok(batch(pcb, bs))
    [Bo6]  cp1(sp(pcb(board1, emp, emp, emp, emp, emp, emp))) = pcb-half(A)
    [Bo7]  cp1(sp(B-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))))
           = pcb-fin(B)
    [Bo8]  cp2(pcb(board1, yellow, red, emp, red, emp, yellow)) = pcb-fin(A)
    [Bo9]  cp2(sp(C-to-be(pcb(board2, emp, emp, emp, emp, emp, emp))))
           = pcb-half(C)
    [Bo10] cp2(sp(rc(pcb(board2, yellow, emp, yellow, emp, yellow, yellow))))
           = pcb-fin(C)
end Batch-operations
```

```
process module Station-actions
begin
  exports
    begin
      atoms
        print-screen: BATCH-ID
        placement-action-1: BATCH-ID
        placement-action-2: BATCH-ID
        reflow-and-clean: BATCH-ID
        test-and-repair: BATCH-ID
    end
  imports
    Batch-ids
end Station-actions
```

```
data module Ports
begin
  exports
    begin
      sorts
        DATA-PORT, BATCH-PORT
      functions
        dp1-0:    ->   DATA-PORT         bp1-0:    ->   BATCH-PORT
        dp1-1:    ->   DATA-PORT         bp1-1:    ->   BATCH-PORT
        dp1-2:    ->   DATA-PORT         bp1-2:    ->   BATCH-PORT
        dp1-3:    ->   DATA-PORT         bp1-3:    ->   BATCH-PORT
        dp1-4:    ->   DATA-PORT         bp1-4:    ->   BATCH-PORT
        dp1-5:    ->   DATA-PORT         bp1-5:    ->   BATCH-PORT
        dp1-6:    ->   DATA-PORT         bp1-6:    ->   BATCH-PORT
        dp1-7:    ->   DATA-PORT         bp1-7:    ->   BATCH-PORT
        dp1-8:    ->   DATA-PORT         bp1-8:    ->   BATCH-PORT
        dp1-9:    ->   DATA-PORT         bp2-0:    ->   BATCH-PORT
        dp2-0:    ->   DATA-PORT         bp2-1:    ->   BATCH-PORT
        dp2-1:    ->   DATA-PORT         bp3-0:    ->   BATCH-PORT
        dp2-2:    ->   DATA-PORT         bp3-1:    ->   BATCH-PORT
        dp3-0:    ->   DATA-PORT         bp3-2:    ->   BATCH-PORT
        dp3-1:    ->   DATA-PORT         bp3-3:    ->   BATCH-PORT
        dp3-2:    ->   DATA-PORT         bp3-4:    ->   BATCH-PORT
        dp3-3:    ->   DATA-PORT         bp4-0:    ->   BATCH-PORT
        dp3-4:    ->   DATA-PORT         bp4-1:    ->   BATCH-PORT
        dp3-5:    ->   DATA-PORT         bp4-2:    ->   BATCH-PORT
        dp4-0:    ->   DATA-PORT         bp4-3:    ->   BATCH-PORT
        dp4-1:    ->   DATA-PORT         bp4-4:    ->   BATCH-PORT
        dp4-2:    ->   DATA-PORT         bp4-5:    ->   BATCH-PORT
        dp4-3:    ->   DATA-PORT         bp4-6:    ->   BATCH-PORT
        dp4-4:    ->   DATA-PORT         bp4-7:    ->   BATCH-PORT
        dp4-5:    ->   DATA-PORT         bp4-8:    ->   BATCH-PORT
        dp4-6:    ->   DATA-PORT         bp4-9:    ->   BATCH-PORT
        dp4-7:    ->   DATA-PORT         bp4-10:   ->   BATCH-PORT
        dp4-8:    ->   DATA-PORT         bp5-0:    ->   BATCH-PORT
        dp5-0:    ->   DATA-PORT         bp5-1:    ->   BATCH-PORT
        dp5-1:    ->   DATA-PORT         bp5-2:    ->   BATCH-PORT
        dp5-2:    ->   DATA-PORT         bp5-3:    ->   BATCH-PORT
        dp5-3:    ->   DATA-PORT         bp5-4:    ->   BATCH-PORT
        dp5-4:    ->   DATA-PORT         bp5-5:    ->   BATCH-PORT
        dp5-5:    ->   DATA-PORT         bp5-6:    ->   BATCH-PORT
        dp5-6:    ->   DATA-PORT         bp5-7:    ->   BATCH-PORT
        dp5-7:    ->   DATA-PORT         bp5-8:    ->   BATCH-PORT
        dp5-8:    ->   DATA-PORT         bp5-9:    ->   BATCH-PORT
        dp5-9:    ->   DATA-PORT         bp5-10:   ->   BATCH-PORT
        dp5-10:   ->   DATA-PORT         bp5-11:   ->   BATCH-PORT
        dp5-11:   ->   DATA-PORT         bp5-12:   ->   BATCH-PORT
        dp6-0:    ->   DATA-PORT         bp5-13:   ->   BATCH-PORT
        dp6-1:    ->   DATA-PORT         bp6-0:    ->   BATCH-PORT
        dp6-2:    ->   DATA-PORT         bp6-1:    ->   BATCH-PORT
        dp6-3:    ->   DATA-PORT         bp6-2:    ->   BATCH-PORT
        dp6-4:    ->   DATA-PORT         bp6-3:    ->   BATCH-PORT
        dp7-0:    ->   DATA-PORT         bp6-4:    ->   BATCH-PORT
        dp7-1:    ->   DATA-PORT         bp6-5:    ->   BATCH-PORT
        dp7-2:    ->   DATA-PORT         bp7-0:    ->   BATCH-PORT
        dp7-3:    ->   DATA-PORT         bp7-1:    ->   BATCH-PORT
        dp7-4:    ->   DATA-PORT         bp7-2:    ->   BATCH-PORT
        dp8-0:    ->   DATA-PORT         bp7-3:    ->   BATCH-PORT
        dp8-1:    ->   DATA-PORT         bp8-0:    ->   BATCH-PORT
        dp8-2:    ->   DATA-PORT         bp8-1:    ->   BATCH-PORT
        dp8-3:    ->   DATA-PORT         bp8-2:    ->   BATCH-PORT
        req-0:    ->   DATA-PORT         disc:     ->   BATCH-PORT
        req-1:    ->   DATA-PORT
        req-2:    ->   DATA-PORT
        req-3:    ->   DATA-PORT
        req-4:    ->   DATA-PORT
        req-5:    ->   DATA-PORT
        req-6:    ->   DATA-PORT
        req-7:    ->   DATA-PORT
    end
end Ports
```

```
process module Communication
begin
  exports
    begin
      atoms
        rb: BATCH-PORT # BATCH
        sb: BATCH-PORT # BATCH
        cb: BATCH-PORT # BATCH
        rm: DATA-PORT  # MESSAGE
        sm: DATA-PORT  # MESSAGE
        cm: DATA-PORT  # MESSAGE
        rr: DATA-PORT  # REQUEST
        sr: DATA-PORT  # REQUEST
        cr: DATA-PORT  # REQUEST
        ro: DATA-PORT  # ORDER
    end
  imports
    Ports, Messages
  communications
    rb(bp, b) | sb(bp, b) = cb(bp, b)
      for bp in BATCH-PORT, b in BATCH
    rm(dp, m) | sm(dp, m) = cm(dp, m)
      for dp in DATA-PORT, m in MESSAGE
    rr(dp, r) | sr(dp, r) = cr(dp, r)
      for dp in DATA-PORT, r in REQUEST
end Communication
```

```
data module Process-ids
begin
  exports
    begin
      sorts
        PROCESS-ID
      functions
        pid1:   ->   PROCESS-ID
        pid2:   ->   PROCESS-ID
        pid3:   ->   PROCESS-ID
        pid4:   ->   PROCESS-ID
        pid5:   ->   PROCESS-ID
    end
end Process-ids



process module Belts
begin
  exports
    begin
      processes
        BELT: BATCH-PORT # BATCH-PORT # BATCH-PORT # BATCH-PORT #
              DATA-PORT # DATA-PORT
    end
  imports
    Communication
  atoms
    transport-belt: BATCH-PORT # BATCH-PORT
  variables
    batch-in-m, batch-in-s,
    batch-out-m, batch-out-s:   ->   BATCH-PORT
    contr-in, contr-out:        ->   DATA-PORT
  definitions
    BELT(batch-in-m, batch-in-s, batch-out-m, batch-out-s, contr-in, contr-out)
      = sm(contr-in, av) .
        ( rm(contr-in, go-mm) . sum(b in BATCH, rb(batch-in-m, b) .
          transport-belt(batch-in-m, batch-out-m) . sm(contr-out, ho(bid(b))) .
          sb(batch-out-m, b))
        + rm(contr-in, go-ms) . sum(b in BATCH, rb(batch-in-m, b) .
          transport-belt(batch-in-m, batch-out-s) . sm(contr-out, ho(bid(b))) .
          sb(batch-out-s, b))
        + rm(contr-in, go-sm) . sum(b in BATCH, rb(batch-in-s, b) .
          transport-belt(batch-in-s, batch-out-m) . sm(contr-out, ho(bid(b))) .
          sb(batch-out-m, b))
        + rm(contr-in, go-ss) . sum(b in BATCH, rb(batch-in-s, b) .
          transport-belt(batch-in-s, batch-out-s) . sm(contr-out, ho(bid(b))) .
          sb(batch-out-s, b))
        ) .
    BELT(batch-in-m, batch-in-s, batch-out-m, batch-out-s, contr-in, contr-out)
end Belts



process module Batch-movers
begin
  exports
    begin
      processes
        BATCH-MOVER: BATCH-PORT # BATCH-PORT # BATCH-PORT # BATCH-PORT #
                     DATA-PORT
    end
  imports
    Communication
  atoms
    move-up-batch: BATCH-PORT # BATCH-PORT
    move-dn-batch: BATCH-PORT # BATCH-PORT
  variables
    batch-in-top, batch-in-bot,
    batch-out-top, batch-out-bot:   ->   BATCH-PORT
    contr:                          ->   DATA-PORT
  definitions
    BATCH-MOVER(batch-in-top, batch-in-bot, batch-out-top, batch-out-bot, contr)
      = sm(contr, av) .
        ( rm(contr, go-up) . sum(b in BATCH, rb(batch-in-bot, b) .
          move-up-batch(batch-in-bot, batch-out-top) . sm(contr, ho(bid(b))) .
          sb(batch-out-top, b))
        + rm(contr, go-dn) . sum(b in BATCH, rb(batch-in-top, b) .
          move-dn-batch(batch-in-top, batch-out-bot) . sm(contr, ho(bid(b))) .
          sb(batch-out-bot, b))
        ) .
    BATCH-MOVER(batch-in-top, batch-in-bot, batch-out-top, batch-out-bot, contr)
end Batch-movers



process module Board-movers
begin
  exports
    begin
      processes
        BOARD-MOVER: BATCH-PORT # BATCH-PORT # DATA-PORT
    end
  imports
    Communication
  atoms
    move-pcb: BATCH-PORT # BATCH-PORT
  processes
    MOVE: BATCH-SIZE # BATCH-PORT # BATCH-PORT
  variables
    batch-in, batch-out, bp-in, bp-out:   ->   BATCH-PORT
    contr:                                ->   DATA-PORT
    pcb:                                  ->   PCB
    n:                                    ->   NAT
  definitions
    BOARD-MOVER(batch-in, batch-out, contr)
      = sm(contr, av) . rm(contr, go) . sum(b in BATCH, rb(batch-in, b).
        MOVE(size(b), batch-in, batch-out) .  sm(contr, ho(bid(b))) .
        sb(batch-out, b)) . BOARD-MOVER(batch-in, batch-out, contr)
    MOVE(one, bp-in, bp-out) = move-pcb(bp-in, bp-out)
    MOVE(two, bp-in, bp-out) = move-pcb(bp-in, bp-out) .
                               move-pcb(bp-in, bp-out)
    MOVE(three, bp-in, bp-out) = move-pcb(bp-in, bp-out) .
                                 move-pcb(bp-in, bp-out) .
                                 move-pcb(bp-in, bp-out)
end Board-movers
```

```
process module Workstations
begin
  parameters
    station-id
      begin
        functions
          k:   ->   PROCESS-ID
      end station-id,
    station-result
      begin
        functions
          proc: BATCH   ->   BATCH
      end station-result,
    station-action
      begin
        atoms
          action: BATCH-ID
      end station-action
  exports
    begin
      processes
        STATION: PROCESS-ID # BATCH-PORT # BATCH-PORT # DATA-PORT
    end
  imports
    Communication, Process-ids
  variables
    batch-in, batch-out:   ->   BATCH-PORT
    contr:                 ->   DATA-PORT
  definitions
    STATION(k, batch-in, batch-out, contr)
      = sm(contr, av) . rm(contr, go) . sum(b in BATCH, rb(batch-in, b) .
        action(bid(b)) . sm(contr, ho(bid(proc(b)))) . sb(batch-out, proc(b))) .
        STATION(k, batch-in, batch-out, contr)
end Workstations
```

```
process module Screen-printing-station
begin
  imports
    Workstations
     {station-id bound by
        [k   ->   pid1]
      to Process-ids
      station-result bound by
        [proc   ->   screen-print]
      to Batch-operations
      station-action bound by
        [action   ->   print-screen]
      to Station-actions}
end Screen-printing-station
```

```
process module Placement-station-1
begin
  imports
    Workstations
     {station-id bound by
        [k   ->   pid2]
      to Process-ids
      station-result bound by
        [proc   ->   comp-place-1]
      to Batch-operations
      station-action bound by
        [action   ->   placement-action-1]
      to Station-actions}
end Placement-station-1
```

```
process module Placement-station-2
begin
  imports
    Workstations
     {station-id bound by
        [k   ->   pid3]
      to Process-ids
      station-result bound by
        [proc   ->   comp-place-2]
      to Batch-operations
      station-action bound by
        [action   ->   placement-action-2]
      to Station-actions}
end Placement-station-2
```

```
process module Reflow-clean-station
begin
  imports
    Workstations
     {station-id bound by
        [k   ->   pid4]
      to Process-ids
      station-result bound by
        [proc   ->   reflow-clean]
      to Batch-operations
      station-action bound by
        [action   ->   reflow-and-clean]
      to Station-actions}
end Reflow-clean-station
```

```
process module Test-repair-station
begin
  imports
    Workstations
     {station-id bound by
        [k   ->   pid5]
      to Process-ids
      station-result bound by
        [proc   ->   test-repair]
      to Batch-operations
      station-action bound by
        [action   ->   test-and-repair]
      to Station-actions}
end Test-repair-station
```

```
process module In-process-stores
begin
  exports
    begin
      processes
        IPS: PRODUCT-TYPE # BATCH-PORT # DATA-PORT
    end
  imports
    Communication
  processes
    IPS: PRODUCT-TYPE # BATCH-PORT # DATA-PORT # BATCH-SIZE
  variables
    store-item:   ->   PRODUCT-TYPE
    batch-io:     ->   BATCH-PORT
    contr:        ->   DATA-PORT
    n:            ->   BATCH-SIZE
  definitions
    IPS(store-item, batch-io, contr) = IPS(store-item, batch-io, contr, zero)
    IPS(store-item, batch-io, contr, n)
      = sm(contr, pres(bn(n))) . sum(bs in BATCH-SIZE,
          rm(contr, take(req(store-item, bs))) .
          rb(batch-io, batch(rc(pcb-fin(store-item)), bs)) .
          IPS(store-item, batch-io, contr, nb(add(bn(bs), bn(n))))
        + rm(contr, give(req(store-item, bs))) .
          sm(contr, ho(id(store-item, bs, del))) .
          sb(batch-io, del(batch(rc(pcb-fin(store-item)), bs))) .
          IPS(store-item, batch-io, contr, nb(sub(bn(n), bn(bs)))) )
end In-process-stores
```

```
process module Raw-material-stores
begin
  exports
    begin
      processes
        RMS: BATCH-PORT # DATA-PORT
    end
  imports
    Communication, Loop-values
  processes
    RMS: BATCH-PORT # DATA-PORT # NAT
  variables
    batch-io:   ->   BATCH-PORT
    contr:      ->   DATA-PORT
    n:          ->   NAT
  definitions
    RMS(batch-io, contr) = RMS(batch-io, contr, top)
    RMS(batch-io, contr, 0) = RMS(batch-io, contr, top)
    RMS(batch-io, contr, s(n))
      = sm(contr, pres(s(n))) . sum(bs in BATCH-SIZE,
        sum(pt in PRODUCT-TYPE, rm(contr, give(req(pt, bs))) .
          sm(contr, ho(id(pt, bs, empty))) .
          sb(batch-io, batch(pcb-empty(pt), bs)) ) .
        RMS(batch-io, contr, sub(s(n), bn(bs))) )
end Raw-material-stores
```

```
process module Warehouse
begin
  exports
    begin
      processes
        WAREHOUSE
    end
  imports
    Communication
  processes
    WAREHOUSE: NAT
  variables
    n:   ->   NAT
  definitions
    WAREHOUSE = WAREHOUSE(0)
    WAREHOUSE(n)
      = sm(dp8-2, pres(n)) . sum(bs in BATCH-SIZE,
        sum(pt in PRODUCT-TYPE, rm(dp8-2, take(req(pt, bs))) .
        rb(bp8-2, ok(batch(rc(pcb-fin(pt)), bs)))) . WAREHOUSE(add(n, bn(bs))))
end Warehouse


-- The recursion on WAREHOUSE(n) should of course be
-- WAREHOUSE(add(n, bn(bs))), but for non-deadlocking
-- simulations this is replaced by WAREHOUSE(n).
```

```
process module Second-side-buffer
begin
  exports
    begin
      processes
        SSB
    end
  imports
    Communication
  definitions
    SSB = sm(dp1-3, av) .
          ( rm(dp1-3, reset)
          + rm(dp1-3, go) . sum(bs in BATCH-SIZE,
            rb(bp1-0, batch(rc(pcb-half(C)), bs)) .
            sm(dp1-3, ho(id(C, bs, half))) .
            sb(bp1-3, batch(rc(pcb-half(C)), bs)) )
          ) . SSB
end Second-side-buffer
```

```
process module Second-side-controller
begin
  exports
    begin
      processes
        SECOND-SIDE-CONTROLLER
    end
  imports
    Communication, Loop-set
  processes
    C1-0 C1-1 C1-2 C1-3 C1-4 C1-5 C1-6 C1-7 C1-8 C1-9
  definitions
    SECOND-SIDE-CONTROLLER = (C1-0 || C1-1 || C1-2 || C1-3 || C1-4 ||
                              C1-5 || C1-6 || C1-7 || C1-8 || C1-9)

    C1-0 = sum(bid in BATCH-ID, rm(dp1-0, ho(bid))) .
           rm(dp1-3, av) . sm(dp1-3, go) . C1-0
    C1-1 = sum(bid in BATCH-ID, rm(dp1-1, ho(bid))) .
           rm(dp1-4, av) . sm(dp1-4, go) . C1-1
    C1-2 = sum(bid in BATCH-ID, rm(dp1-2, ho(bid))) .
           rm(dp1-5, av) . sm(dp1-5, go) . C1-2
    C1-3 = sum(bid in BATCH-ID, rm(dp1-3, ho(bid))) .
           rm(dp1-6, av) . sm(dp1-6, go) . C1-3
    C1-4 = sum(bid in BATCH-ID, rm(dp1-4, ho(bid))) .
           rm(dp1-7, av) . sm(dp1-7, go-sm) . C1-4
    C1-5 = sum(bid in BATCH-ID, rm(dp1-5, ho(bid))) .
           rm(dp1-8, av) . sm(dp1-8, go-sm) . C1-5
    C1-6 = sum(bid in BATCH-ID, rm(dp1-6, ho(bid))) .
           rm(dp1-9, av) . sm(dp1-9, go-sm) . C1-6
    C1-7 = sum(bid in BATCH-ID, rm(dp1-7, ho(bid))) .
           rm(dp1-8, av) . sm(dp1-8, go-mm) . C1-7
    C1-8 = sum(bid in BATCH-ID, rm(dp1-8, ho(bid))) .
           rm(dp1-9, av) . sm(dp1-9, go-mm) . C1-8
    C1-9 = sum(bs in BATCH-SIZE,
             rr(req-7, req(A, bs)) . rm(dp1-3, av) . sum(n in LOOP-SET,
             rm(dp1-1, pres(n))) . sm(dp1-1, give(req(A, bs))) .
             sm(dp1-3, reset)
           + rr(req-7, req(B, bs)) . rm(dp1-3, av) . sum(n in LOOP-SET,
             rm(dp1-2, pres(n))) . sm(dp1-2, give(req(B, bs))) .
             sm(dp1-3, reset)
           + rr(req-7, req(C, bs)) . rm(dp1-3, av) . sum(n in LOOP-SET,
             rm(dp1-2, pres(n))) . sm(dp1-2, give(req(C, bs))) .
             sm(dp1-3, reset)
           ) . C1-9
end Second-side-controller
```

```
process module Screen-print-controller
begin
  exports
    begin
      processes
        SCREEN-PRINT-CONTROLLER
    end
  imports
    Communication
  processes
    C2-0 C2-1 C2-2
  definitions
    SCREEN-PRINT-CONTROLLER = (C2-0 || C2-1 || C2-2)
    C2-0 = sum(bid in BATCH-ID, rm(dp2-0, ho(bid))) .
           rm(dp2-1, av) . sm(dp2-1, go) . C2-0
    C2-1 = sum(bs in BATCH-SIZE,
             (rm(dp2-1, ho(id(A, bs, empty))) +
             rm(dp2-1, ho(id(B, bs, empty)))) .
             rm(dp2-2, av) . sm(dp2-2, go-ms)
           + (rm(dp2-1, ho(id(C, bs, empty))) +
             rm(dp2-1, ho(id(C, bs, half)))) .
             rm(dp2-2, av) . sm(dp2-2, go-mm)
           ) . C2-1
    C2-2 = sum(req in REQUEST, rr(req-6, req) . sr(req-7, req)) . C2-2
end Screen-print-controller
```

```
process module Placement-controller-1
begin
  exports
    begin
      processes
        PLACEMENT-CONTROLLER-1
    end
  imports
    Communication
  processes
    C3-0 C3-1 C3-2 C3-3 C3-4
  definitions
    PLACEMENT-CONTROLLER-1 = (C3-0 || C3-1 || C3-2 || C3-3 || C3-4)
    C3-0 = sum(bs in BATCH-SIZE,
             (rm(dp3-0, ho(id(A, bs, empty))) +
             rm(dp3-0, ho(id(B, bs, empty)))) .
             rm(dp3-2, av) . sm(dp3-2, go)
           + (rm(dp3-0, ho(id(C, bs, empty))) +
             rm(dp3-0, ho(id(C, bs, half)))) .
             rm(dp3-1, av) . sm(dp3-1, go-ms)
           ) . C3-0
    C3-1 = sum(bid in BATCH-ID, rm(dp3-2, ho(bid))) .
           rm(dp3-3, av) . sm(dp3-3, go-sm) . C3-1
    C3-2 = sum(bid in BATCH-ID, rm(dp3-3, ho(bid))) .
           rm(dp3-4, av) . sm(dp3-4, go) . C3-2
    C3-3 = sum( bs in BATCH-SIZE,
             rm(dp3-4, ho(id(A, bs, half))) . rm(dp3-5, av) . sm(dp3-5, go-mm)
           + rm(dp3-4, ho(id(B, bs, fin))) . rm(dp3-5, av) . sm(dp3-5, go-ms)
           ) . C3-3
    C3-4 = sum(req in REQUEST, rr(req-5, req) . sr(req-6, req)) . C3-4
end Placement-controller-1
```

```
process module Placement-controller-2
begin
  exports
    begin
      processes
        PLACEMENT-CONTROLLER-2
    end
  imports
    Communication
  processes
    C4-0 C4-1 C4-2 C4-3 C4-4 C4-5 C4-6 C4-7 C4-8
  definitions
    PLACEMENT-CONTROLLER-2
      = (C4-0 || C4-1 || C4-2 || C4-3 || C4-4 || C4-5 || C4-6 || C4-7 || C4-8)
    C4-0 = sum(bid in BATCH-ID, rm(dp4-0, ho(bid))) .
           rm(dp4-2, av) . sm(dp4-2, go-dn) . C4-0
    C4-1 = sum(bs in BATCH-SIZE,
             rm(dp4-1, ho(id(A, bs, half))) . rm(dp4-4, av) . sm(dp4-4, go-mm)
           + rm(dp4-1, ho(id(B, bs, fin))) . rm(dp4-2, av) . sm(dp4-2, go-up)
           ) . C4-1
    C4-2 = sum(bs in BATCH-SIZE,
             rm(dp4-2, ho(id(B, bs, fin))) . rm(dp4-3, av) . sm(dp4-3, go-sm)
           + (rm(dp4-2, ho(id(C, bs, empty))) +
             rm(dp4-2, ho(id(C, bs, half)))) .
             rm(dp4-4, av) . sm(dp4-4, go-sm)
           ) . C4-2
    C4-3 = sum(bid in BATCH-ID, rm(dp4-3, ho(bid))) .
           rm(dp4-8, av) . sm(dp4-8, go-mm) . C4-3
    C4-4 = sum(bid in BATCH-ID, rm(dp4-4, ho(bid))) .
           rm(dp4-5, av) . sm(dp4-5, go) . C4-4
    C4-5 = sum(bid in BATCH-ID, rm(dp4-5, ho(bid))) .
           rm(dp4-6, av) . sm(dp4-6, go-ms) . C4-5
    C4-6 = sum(bid in BATCH-ID, rm(dp4-6, ho(bid))) .
           rm(dp4-7, av) . sm(dp4-7, go) . C4-6
    C4-7 = sum(bid in BATCH-ID, rm(dp4-7, ho(bid))) .
           rm(dp4-8, av) . sm(dp4-8, go-sm) . C4-7
    C4-8 = sum(req in REQUEST, rr(req-4, req) . sr(req-5, req)) . C4-8
end Placement-controller-2


process module Ips-controller
begin
  exports
    begin
      processes
        IPS-CONTROLLER
    end
  imports
    Communication, Loop-set
  processes
    C5-0 C5-1 C5-2 C5-3 C5-4 C5-5 C5-6 C5-7 C5-8 C5-9 C5-10 C5-11 C5-12 C5-13
  definitions
    IPS-CONTROLLER =  sr(req-4, req(A, three)) .
                      sr(req-4, req(B, three)) .
                      sr(req-4, req(C, three)) .
                     (C5-0 || C5-1 || C5-2 || C5-3  || C5-4  || C5-5  || C5-6 ||
                      C5-7 || C5-8 || C5-9 || C5-10 || C5-11 || C5-12 || C5-13)

    C5-0  = sum(bid in BATCH-ID, rm(dp5-0, ho(bid))) .
            rm(dp5-1, av) . sm(dp5-1, go) . C5-0
    C5-1  = sum(bs in BATCH-SIZE,
              rm(dp5-1, ho(id(A, bs, fin))) . rm(dp5-2, av) . sm(dp5-2, go-ms)
            + (rm(dp5-1, ho(id(B, bs, fin))) + rm(dp5-1, ho(id(C, bs, fin))) +
              rm(dp5-1, ho(id(C, bs, half)))) . rm(dp5-2, av) . sm(dp5-2, go-mm)
            ) . C5-1
    C5-2  = sum(bs in BATCH-SIZE,
              rm(dp5-2, ho(id(A, bs, fin))) . rm(dp5-3, av) . sm(dp5-3, go-dn)
            + rm(dp5-2, ho(id(B, bs, fin))) . rm(dp5-5, av) . sm(dp5-5, go-ms)
            + (rm(dp5-2, ho(id(C, bs, fin))) + rm(dp5-2, ho(id(C, bs, half)))) .
              rm(dp5-5, av) . sm(dp5-5, go-mm)
            ) . C5-2
    C5-3  = sum(bs in BATCH-SIZE,
              rm(dp5-3, ho(id(A, bs, fin))) . sum(n in LOOP-SET,
              rm(dp5-4, pres(n))) . sm(dp5-4, take(req(A, bs)))
            + rm(dp5-3, ho(id(A, bs, del))) . rm(dp5-5, av) . sm(dp5-5, go-sm)
            ) . C5-3
    C5-4  = sum(bs in BATCH-SIZE, rm(dp5-4, ho(id(A, bs, del)))) .
            rm(dp5-3, av) . sm(dp5-3, go-up) . C5-4
    C5-5  = sum(bs in BATCH-SIZE,
              rm(dp5-5, ho(id(B, bs, fin))) . rm(dp5-6, av) . sm(dp5-6, go-dn)
            + rm(dp5-5, ho(id(C, bs, fin))) . rm(dp5-8, av) . sm(dp5-8, go-ms)
            + (rm(dp5-5, ho(id(A, bs, del))) + rm(dp5-5, ho(id(C, bs, half)))) .
              rm(dp5-8, av) . sm(dp5-8, go-mm)
            ) . C5-5
    C5-6  = sum(bs in BATCH-SIZE,
              rm(dp5-6, ho(id(B, bs, fin))) . sum(n in LOOP-SET,
              rm(dp5-7, pres(n))) . sm(dp5-7, take(req(B, bs)))
            + rm(dp5-6, ho(id(B, bs, del))) . rm(dp5-8, av) . sm(dp5-8, go-sm)
            ) . C5-6
    C5-7  = sum(bs in BATCH-SIZE, rm(dp5-7, ho(id(B, bs, del)))) .
            rm(dp5-6, av) . sm(dp5-6, go-up) . C5-7
    C5-8  = sum(bs in BATCH-SIZE,
              rm(dp5-8, ho(id(C, bs, fin))) . rm(dp5-9, av) . sm(dp5-9, go-dn)
            + rm(dp5-8, ho(id(C, bs, half))) . rm(dp5-11, av) .
              sm(dp5-11, go-ms)
            + (rm(dp5-8, ho(id(A, bs, del))) + rm(dp5-8, ho(id(B, bs, del)))) .
              rm(dp5-11, av) . sm(dp5-11, go-mm)
            ) . C5-8
    C5-9  = sum(bs in BATCH-SIZE,
              rm(dp5-9, ho(id(C, bs, fin))) . sum(n in LOOP-SET,
              rm(dp5-10, pres(n))) . sm(dp5-10, take(req(C, bs)))
            + rm(dp5-9, ho(id(C, bs, del))) . rm(dp5-11, av) . sm(dp5-11, go-sm)
            ) . C5-9
    C5-10 = sum(bs in BATCH-SIZE, rm(dp5-10, ho(id(C, bs, del)))) .
            rm(dp5-9, av) . sm(dp5-9, go-up) . C5-10
    C5-11 = sum(bs in BATCH-SIZE, rr(req-3, req(A, bs)) .
            (rm(dp5-4, pres(bn(bs))) + rm(dp5-4, pres(s(bn(bs)))) +
            rm(dp5-4, pres(s(s(bn(bs)))))) . sm(dp5-4, give(req(A, bs))) .
            sr(req-4, req(A, bs)) ) . C5-11
    C5-12 = sum(bs in BATCH-SIZE, rr(req-3, req(B, bs)) .
            (rm(dp5-7, pres(bn(bs))) + rm(dp5-7, pres(s(bn(bs)))) +
            rm(dp5-7, pres(s(s(bn(bs)))))) . sm(dp5-7, give(req(B, bs))) .
            sr(req-4, req(B, bs)) ) . C5-12
    C5-13 = sum(bs in BATCH-SIZE, rr(req-3, req(C, bs)) .
            (rm(dp5-10, pres(bn(bs))) + rm(dp5-10, pres(s(bn(bs)))) +
            rm(dp5-10, pres(s(s(bn(bs)))))) . sm(dp5-10, give(req(C, bs))) .
            sr(req-4, req(C, bs)) ) . C5-13
end Ips-controller
```

```
process module Bottleneck-controller
begin
  exports
    begin
      processes
        BOTTLENECK-CONTROLLER
    end
  imports
    Communication
  processes
    C6-0 C6-1 C6-2 C6-3
  definitions
    BOTTLENECK-CONTROLLER = (C6-0 || C6-1 || C6-2 || C6-3)
    C6-0 = sum(bs in BATCH-SIZE,
             rm(dp6-0, ho(id(C, bs, half))) . rm(dp6-2, av) . sm(dp6-2, go-up)
           + sum(pt in PRODUCT-TYPE, rm(dp6-0, ho(id(pt, bs, del)))) .
             rm(dp6-3, av) . sm(dp6-3, go-mm)) . C6-0
    C6-1 = sum(bs in BATCH-SIZE, sum(pt in PRODUCT-TYPE,
             rm(dp6-1, ho(id(pt, bs, rep))) . rm(dp6-2, av) . sm(dp6-2, go-dn)
           + rm(dp6-1, ho(id(pt, bs, ok))) . rm(dp6-4, av) . sm(dp6-4, go-ms)
           )) . C6-1
    C6-2 = sum(bs in BATCH-SIZE,
             rm(dp6-2, ho(id(C, bs, half))) . rm(dp6-4, av) . sm(dp6-4, go-sm)
           + sum(pt in PRODUCT-TYPE, rm(dp6-2, ho(id(pt, bs, rep)))) .
             rm(dp6-3, av) . sm(dp6-3, go-sm)) . C6-2
    C6-3 = sum(req in REQUEST, rr(req-2, req) . sr(req-3, req)) . C6-3
end Bottleneck-controller


process module Test-repair-controller
begin
  exports
    begin
      processes
        TEST-REPAIR-CONTROLLER
    end
  imports
    Communication
  processes
    C7-0 C7-1 C7-2 C7-3 C7-4
  definitions
    TEST-REPAIR-CONTROLLER = (C7-0 || C7-1 || C7-2 || C7-3 || C7-4)
    C7-0 = sum(bid in BATCH-ID, rm(dp7-0, ho(bid))) .
           rm(dp7-1, av) . sm(dp7-1, go) . C7-0
    C7-1 = sum(bid in BATCH-ID, rm(dp7-1, ho(bid))) .
           rm(dp7-2, av) . sm(dp7-2, go-ms) . C7-1
    C7-2 = sum(bid in BATCH-ID, rm(dp7-2, ho(bid))) .
           rm(dp7-3, av) . sm(dp7-3, go-up) . C7-2
    C7-3 = sum(bs in BATCH-SIZE, sum(pt in PRODUCT-TYPE,
             rm(dp7-3, ho(id(pt, bs, rep))) . rm(dp7-4, av) . sm(dp7-4, go-ss)
           + rm(dp7-3, ho(id(pt, bs, ok))) . rm(dp7-4, av) . sm(dp7-4, go-sm)
           )) . C7-3
    C7-4 = sum(req in REQUEST, rr(req-1, req) . sr(req-2, req)) . C7-4
end Test-repair-controller


process module Warehouse-controller
begin
  exports
    begin
      processes
        WAREHOUSE-CONTROLLER
    end
  imports
    Communication, Loop-set
  processes
    C8-0 C8-1 C8-2
    HANDLE: PRODUCT-TYPE # NAT
  variables
    n:    ->   NAT
    pt:   ->   PRODUCT-TYPE
  definitions
    WAREHOUSE-CONTROLLER = (C8-0 || C8-1 || C8-2)
    C8-0 = sum(bs in BATCH-SIZE, sum(pt in PRODUCT-TYPE,
               rm(dp8-0, ho(id(pt, bs, ok))) ) . rm(dp8-1, av) . sm(dp8-1, go)
             + rm(dp8-0, ho(id(C, bs, half))) . rm(dp8-3, av) . sm(dp8-3, go-ms)
             ) . C8-0
    C8-1 = sum(bs in BATCH-SIZE, sum(pt in PRODUCT-TYPE,
             rm(dp8-1, ho(id(pt, bs, ok))) . sum(n in LOOP-SET,
             rm(dp8-2, pres(n))) . sm(dp8-2, take(req(pt, bs))))) . C8-1
    C8-2 = sum(n in LOOP-SET, sum(pt in PRODUCT-TYPE,
             ro(req-0, ord(pt, n)) . HANDLE(pt, n))) . C8-2
    HANDLE(pt, 0)          = skip
    HANDLE(pt, s(0))       = sr(req-1, req(pt, nb(s(0))))
    HANDLE(pt, s(s(0)))    = sr(req-1, req(pt, nb(s(s(0)))))
    HANDLE(pt, s(s(s(n)))) = sr(req-1, req(pt, nb(s(s(s(0)))))) . HANDLE(pt, n)
end Warehouse-controller


process module Controll-zone-1
begin
  exports
    begin
      processes
        CZ-1
    end
  imports
    Raw-material-stores, Second-side-buffer,
    Belts, Board-movers, Second-side-controller
  processes
    RMS-1 RMS-2 BELT-1 BELT-2 BELT-3 BoM-1 BoM-2 BoM-3
  sets
    of BATCH-PORT
      BP-1 = { bp1-1, bp1-2, bp1-3, bp1-4, bp1-5, bp1-6, bp1-7, bp1-8, disc }
    of DATA-PORT
      DP-1 = { dp1-1, dp1-2, dp1-3, dp1-4, dp1-5, dp1-6, dp1-7, dp1-8, dp1-9 }
    of atoms
      H-1 =   {rb(bp, b), sb(bp, b) | bp in BP-1, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-1, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-1, r in REQUEST}
  definitions
    CZ-1 = encaps(H-1, RMS-1 || RMS-2 || SSB || BELT-1 || BELT-2 || BELT-3 ||
                       BoM-1 || BoM-2 || BoM-3 || SECOND-SIDE-CONTROLLER)
    RMS-1 = RMS(bp1-1, dp1-1)
    RMS-2 = RMS(bp1-2, dp1-2)
    BELT-1 = BELT(disc, bp1-4, bp1-5, disc, dp1-7, dp1-7)
    BELT-2 = BELT(bp1-5, bp1-6, bp1-7, disc, dp1-8, dp1-8)
    BELT-3 = BELT(bp1-7, bp1-8, bp2-0, disc, dp1-9, dp2-0)
    BoM-1 = BOARD-MOVER(bp1-1, bp1-4, dp1-4)
    BoM-2 = BOARD-MOVER(bp1-2, bp1-6, dp1-5)
    BoM-3 = BOARD-MOVER(bp1-3, bp1-8, dp1-6)
end Controll-zone-1
```

```
process module Controll-zone-2
begin
  exports
    begin
      processes
        CZ-2
    end
  imports
    Screen-printing-station, Belts, Screen-print-controller
  processes
    SP BELT-4
  sets
    of BATCH-PORT
      BP-2 = { bp2-1, disc }
    of DATA-PORT
      DP-2 = { dp2-1, dp2-2 }
    of atoms
      H-2 =   {rb(bp, b), sb(bp, b) | bp in BP-2, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-2, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-2, r in REQUEST}
  definitions
    CZ-2 = encaps(H-2, SP || BELT-4 || SCREEN-PRINT-CONTROLLER)
    SP = STATION(pid1, bp2-0, bp2-1, dp2-1)
    BELT-4 = BELT(bp2-1, disc, bp3-0, bp3-1, dp2-2, dp3-0)
end Controll-zone-2

process module Controll-zone-3
begin
  exports
    begin
      processes
        CZ-3
    end
  imports
    Placement-station-1, Belts, Board-movers, Placement-controller-1
  processes
    CP1 BELT-5 BELT-6 BELT-7 BoM-4
  sets
    of BATCH-PORT
      BP-3 = { bp3-2, bp3-3, bp3-4, disc }
    of DATA-PORT
      DP-3 = { dp3-1, dp3-2, dp3-3, dp3-4, dp3-5 }
    of atoms
      H-3 =   {rb(bp, b), sb(bp, b) | bp in BP-3, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-3, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-3, r in REQUEST}
  definitions
    CZ-3 = encaps(H-3, CP1 || BELT-5 || BELT-6 || BELT-7 || BoM-4 ||
                       PLACEMENT-CONTROLLER-1)
    CP1 = STATION(pid2, bp3-3, bp3-4, dp3-4)
    BELT-5 = BELT(bp3-0, disc, bp4-0, bp4-2, dp3-1, dp4-0)
    BELT-6 = BELT(disc, bp3-2, bp3-3, disc, dp3-3, dp3-3)
    BELT-7 = BELT(bp3-4, disc, bp4-1, bp4-3, dp3-5, dp4-1)
    BoM-4 = BOARD-MOVER(bp3-1, bp3-2, dp3-2)
end Controll-zone-3

process module Controll-zone-4
begin
  exports
    begin
      processes
        CZ-4
    end
  imports
    Placement-station-2, Belts, Batch-movers, Board-movers, Placement-controller-2
  processes
    CP2 BELT-8 BELT-9 BELT-10 BELT-11 BaM-1 BoM-5
  sets
    of BATCH-PORT
      BP-4 = { bp4-4, bp4-5, bp4-6, bp4-7, bp4-8, bp4-9, bp4-10, disc }
    of DATA-PORT
      DP-4 = { dp4-2, dp4-3, dp4-4, dp4-5, dp4-6, dp4-7, dp4-8 }
    of atoms
      H-4 =   {rb(bp, b), sb(bp, b) | bp in BP-4, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-4, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-4, r in REQUEST}
  definitions
    CZ-4 = encaps(H-4, CP2 || BELT-8 || BELT-9 || BELT-10 || BELT-11 ||
                       BaM-1 || BoM-5 || PLACEMENT-CONTROLLER-2)
    CP2 = STATION(pid3, bp4-6, bp4-7, dp4-5)
    BELT-8 = BELT(bp4-0, bp4-5, bp4-9, disc, dp4-3, dp4-3)
    BELT-9 = BELT(bp4-1, bp4-4, bp4-6, disc, dp4-4, dp4-4)
    BELT-10 = BELT(bp4-7, disc, disc, bp4-8, dp4-6, dp4-6)
    BELT-11 = BELT(bp4-9, bp4-10, bp5-0, disc, dp4-8, dp5-0)
    BaM-1 = BATCH-MOVER(bp4-2, bp4-3, bp4-5, bp4-4, dp4-2)
    BoM-5 = BOARD-MOVER(bp4-8, bp4-10, dp4-7)
end Controll-zone-4

process module Controll-zone-5
begin
  exports
    begin
      processes
        CZ-5
    end
  imports
    Reflow-clean-station, In-process-stores, Belts, Batch-movers, Ips-controller
  processes
    RC IPS-1 IPS-2 IPS-3 BELT-12 BELT-13 BELT-14 BELT-15 BaM-2 BaM-3 BaM-4
  sets
    of BATCH-PORT
      BP-5 = { bp5-1, bp5-2, bp5-3, bp5-4, bp5-5, bp5-6, bp5-7, bp5-8,
               bp5-9, bp5-10, bp5-11, bp5-12, bp5-13, disc }
    of DATA-PORT
      DP-5 = { dp5-1, dp5-2, dp5-3, dp5-4, dp5-5, dp5-6,
               dp5-7, dp5-8, dp5-9, dp5-10, dp5-11 }
    of atoms
      H-5 =   {rb(bp, b), sb(bp, b) | bp in BP-5, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-5, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-5, r in REQUEST}
  definitions
    CZ-5 = encaps(H-5, RC || IPS-1 || IPS-2 || IPS-3 || BELT-12 || BELT-13 ||
                  BELT-14 || BELT-15 || BaM-2 || BaM-3 || BaM-4 || IPS-CONTROLLER)
    RC = STATION(pid4, bp5-0, bp5-1, dp5-1)
    IPS-1 = IPS(A, bp5-4, dp5-4)            IPS-2 = IPS(B, bp5-8, dp5-7)
    IPS-3 = IPS(C, bp5-12, dp5-10)
    BELT-12 = BELT(bp5-1, disc, bp5-2, bp5-3, dp5-2, dp5-2)
    BELT-13 = BELT(bp5-2, bp5-5, bp5-6, bp5-7, dp5-5, dp5-5)
    BELT-14 = BELT(bp5-6, bp5-9, bp5-10, bp5-11, dp5-8, dp5-8)
    BELT-15 = BELT(bp5-10, bp5-13, bp6-0, bp6-2, dp5-11, dp6-0)
    BaM-2 = BATCH-MOVER(bp5-3, bp5-4, bp5-5, bp5-4, dp5-3)
    BaM-3 = BATCH-MOVER(bp5-7, bp5-8, bp5-9, bp5-8, dp5-6)
    BaM-4 = BATCH-MOVER(bp5-11, bp5-12, bp5-13, bp5-12, dp5-9)
end Controll-zone-5
```

```
process module Controll-zone-6
begin
  exports
    begin
      processes
        CZ-6
    end
  imports
    Belts, Batch-movers, Bottleneck-controller
  processes
    BELT-16 BELT-19 BaM-5
  sets
    of BATCH-PORT
      BP-6 = { bp6-4, bp6-5, disc }
    of DATA-PORT
      DP-6 = { dp6-2, dp6-3, dp6-4 }
    of atoms
      H-6 =   {rb(bp, b), sb(bp, b) | bp in BP-6, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-6, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-6, r in REQUEST}
  definitions
    CZ-6 = encaps(H-6, BELT-16 || BELT-19 || BaM-5 || BOTTLENECK-CONTROLLER)

    BELT-16 = BELT(bp6-0, bp6-4, bp7-0, disc, dp6-3, dp7-0)
    BELT-19 = BELT(bp6-1, bp6-5, bp8-0, bp8-1, dp6-4, dp8-0)
    BaM-5 = BATCH-MOVER(bp6-3, bp6-2, bp6-5, bp6-4, dp6-2)
end Controll-zone-6
```

```
process module Controll-zone-7
begin
  exports
    begin
      processes
        CZ-7
    end
  imports
    Test-repair-station, Belts, Batch-movers, Test-repair-controller
  processes
    TR BELT-17 BELT-18 BaM-6
  sets
    of BATCH-PORT
      BP-7 = { bp7-1, bp7-2, bp7-3, disc }
    of DATA-PORT
      DP-7 = { dp7-1, dp7-2, dp7-3, dp7-4 }
    of atoms
      H-7 =   {rb(bp, b), sb(bp, b) | bp in BP-7, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-7, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-7, r in REQUEST}
  definitions
    CZ-7 = encaps(H-7, TR || BELT-17 || BELT-18 || BaM-6 ||
                       TEST-REPAIR-CONTROLLER)

    TR = STATION(pid5, bp7-0, bp7-1, dp7-1)
    BELT-17 = BELT(bp7-1, disc, disc, bp7-2, dp7-2, dp7-2)
    BELT-18 = BELT(disc, bp7-3, bp6-1, bp6-3, dp7-4, dp6-1)
    BaM-6 = BATCH-MOVER(disc, bp7-2, bp7-3, disc, dp7-3)
end Controll-zone-7
```

```
process module Controll-zone-8
begin
  exports
    begin
      processes
        CZ-8
    end
  imports
    Warehouse, Belts, Board-movers, Warehouse-controller
  processes
    BELT-20 BoM-6
  sets
    of BATCH-PORT
      BP-8 = { bp8-2, disc }
    of DATA-PORT
      DP-8 = { dp8-1, dp8-2, dp8-3 }
    of atoms
      H-8 =   {rb(bp, b), sb(bp, b) | bp in BP-8, b in BATCH}
            + {rm(dp, m), sm(dp, m) | dp in DP-8, m in MESSAGE}
            + {rr(dp, r), sr(dp, r) | dp in DP-8, r in REQUEST}
  definitions
    CZ-8 = encaps(H-8, WAREHOUSE || BELT-20 || BoM-6 ||
                       WAREHOUSE-CONTROLLER)

    BELT-20 = BELT(bp8-0, disc, disc, bp1-0, dp8-3, dp1-0)
    BoM-6 = BOARD-MOVER(bp8-1, bp8-2, dp8-1)
end Controll-zone-8
```

```
process module Model-factory
begin
  exports
    begin
      processes
        MOD-FACT
    end
  imports
    Controll-zone-1, Controll-zone-2, Controll-zone-3, Controll-zone-4,
    Controll-zone-5, Controll-zone-6, Controll-zone-7, Controll-zone-8
  sets
    of BATCH-PORT
      BP = {bp1-0, bp2-0, bp3-0, bp3-1, bp4-0, bp4-1, bp4-2, bp4-3,
            bp5-0, bp6-0, bp6-1, bp6-2, bp6-3, bp7-0, bp8-0, bp8-1}
    of DATA-PORT
      DP = {dp1-0, dp2-0, dp3-0, dp4-0, dp4-1, dp5-0, dp6-0, dp6-1, dp7-0,
            dp8-0, req-1, req-2, req-3, req-4, req-5, req-6, req-7}
    of atoms
      H =   {rb(bp, b), sb(bp, b) | bp in BP, b in BATCH}
          + {rm(dp, m), sm(dp, m) | dp in DP, m in MESSAGE}
          + {rr(dp, r), sr(dp, r) | dp in DP, r in REQUEST}
  definitions
    MOD-FACT
      = encaps(H, CZ-1 || CZ-2 || CZ-3 || CZ-4 ||
                  CZ-5 || CZ-6 || CZ-7 || CZ-8)
end Model-factory
```